Cortex-M structure

CMSIS = Cortex Microcontroller Software Interface Standard
Cortex CPU core registers

- Two processor modes:
  - Thread mode for User tasks
  - Handler mode for O/S tasks and exceptions
- Stack-based exception model
- Vector table contains addresses

**Process SP** (handler or thread mode – select in CONTROL reg.)

**Main SP** (selected at reset – always used in handler mode)

**Convention:**
- PSP in thread mode,
- MSP in O/S & handler mode
Cortex-M4 processor operating modes

- **Thread** mode – normal processing
- **Handler** mode – interrupt/exception processing
- Privilege levels = **User** and **Privileged**
  - Supports basic “security” & memory access protection
  - Supervisor/operating system usually privileged
Cortex-M4 interrupts/exceptions

- Interrupts/exceptions managed by Nested Vectored Interrupt Controller (NVIC)
- CPU state/context (subset of registers) saved on the stack
  
  R0-R3, R12, LR, PC, PSR

- PC loaded from a vector table, located at 0x0000_0000
  - Vector fetched (Flash memory) while saving state (SRAM)
  - Typical latency = 12 cycles
Exception states

- Each exception is in one of the following states:
  - **Inactive**: The exception is not active and not pending.
  - **Pending**: The exception is waiting to be serviced by the processor.
  - **Active**: The exception is being serviced by the processor but has not completed.
  - **Active and pending**: The exception is being serviced by the processor and there is a pending exception from the same source.

- An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
- An exception handler can interrupt (preempt) the execution of another exception handler. In this case both exceptions are in the active state.
## Cortex-M CPU and peripheral exceptions

<table>
<thead>
<tr>
<th></th>
<th>Priority¹</th>
<th>IRQ#²</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>-3</td>
<td></td>
<td>Power-up or warm reset</td>
</tr>
<tr>
<td>NMI</td>
<td>-2</td>
<td>-14</td>
<td>Non-maskable interrupt from peripheral or software</td>
</tr>
<tr>
<td>HardFault</td>
<td>-1</td>
<td>-13</td>
<td>Error during exception processing or no other handler</td>
</tr>
<tr>
<td>MemManage</td>
<td>Config</td>
<td>-12</td>
<td>Memory protection fault (MPU-detected)</td>
</tr>
<tr>
<td>BusFault</td>
<td>Config</td>
<td>-11</td>
<td>AHB data/prefetch aborts</td>
</tr>
<tr>
<td>UsageFault</td>
<td>Config</td>
<td>-10</td>
<td>Instruction execution fault - undefined instruction, illegal unaligned access</td>
</tr>
<tr>
<td>SVCcall</td>
<td>Config</td>
<td>-5</td>
<td>System service call (SVC) instruction</td>
</tr>
<tr>
<td>DebugMonitor</td>
<td>Config</td>
<td></td>
<td>Break points/watch points/etc.</td>
</tr>
<tr>
<td>PendSV</td>
<td>Config</td>
<td>-2</td>
<td>Interrupt-driven request for system service</td>
</tr>
<tr>
<td>SysTick</td>
<td>Config</td>
<td>-1</td>
<td>System tick timer reaches 0</td>
</tr>
<tr>
<td>IRQ0</td>
<td>Config</td>
<td>0</td>
<td>Signaled by peripheral or by software request</td>
</tr>
<tr>
<td>IRQ1 (etc.)</td>
<td>Config</td>
<td>1</td>
<td>Signaled by peripheral or by software request</td>
</tr>
</tbody>
</table>

¹ Lowest priority # = highest priority
² IRQ# used in CMSIS function calls

### Vendor peripheral interrupts
-IRQ0 .. IRQ44
Vector table

- 32-bit vector (handler address) loaded into PC, while saving CPU context.
- Reset vector includes initial stack pointer
- Peripherals use positive IRQ #s
- CPU exceptions use negative IRQ #s
- IRQ # used in CMSIS function calls
- Cortex-M4 allows up to 240 IRQs
- IRQ priorities user-programmable
- NMI & HardFault priorities fixed

<table>
<thead>
<tr>
<th>Exception number</th>
<th>IRQ number</th>
<th>Offset</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>16+n</td>
<td>0x0040+4n</td>
<td></td>
<td>IRQ2</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>0x004C</td>
<td>IRQ1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0x0048</td>
<td>IRQ0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0x0044</td>
<td>Systick</td>
</tr>
<tr>
<td>15</td>
<td>-1</td>
<td>0x0040</td>
<td>PendSV</td>
</tr>
<tr>
<td>14</td>
<td>-2</td>
<td>0x003C</td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>-3</td>
<td>0x0038</td>
<td>Reserved for Debug</td>
</tr>
<tr>
<td>12</td>
<td>-4</td>
<td>0x002C</td>
<td>SVCALL</td>
</tr>
<tr>
<td>11</td>
<td>-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-6</td>
<td>0x0018</td>
<td>Usage fault</td>
</tr>
<tr>
<td>9</td>
<td>-7</td>
<td>0x0014</td>
<td>Bus fault</td>
</tr>
<tr>
<td>8</td>
<td>-8</td>
<td>0x0010</td>
<td>Memory management fault</td>
</tr>
<tr>
<td>7</td>
<td>-9</td>
<td>0x000C</td>
<td>Hard fault</td>
</tr>
<tr>
<td>6</td>
<td>-10</td>
<td>0x0008</td>
<td>NMI</td>
</tr>
<tr>
<td>5</td>
<td>-11</td>
<td>0x0004</td>
<td>Reset</td>
</tr>
<tr>
<td>4</td>
<td>-12</td>
<td>0x0000</td>
<td>Initial SP value</td>
</tr>
<tr>
<td>3</td>
<td>-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-16</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>Priority</td>
<td>Type of priority</td>
<td>Acronym</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>-3</td>
<td>fixed</td>
<td>Reset</td>
<td>Reset</td>
</tr>
<tr>
<td>6</td>
<td>settable</td>
<td>SysTick</td>
<td>System tick timer</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>settable</td>
<td>WWDG</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>settable</td>
<td>PVD</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>settable</td>
<td>TAMP_STAMP</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>settable</td>
<td>RTC_WKUP</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>settable</td>
<td>FLASH</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>settable</td>
<td>RCC</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>settable</td>
<td>EXTI0</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>settable</td>
<td>EXTI1</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>settable</td>
<td>EXTI2</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>settable</td>
<td>EXTI3</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>settable</td>
<td>EXTI4</td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>settable</td>
<td>DMA1_Stream0</td>
</tr>
<tr>
<td>12</td>
<td>19</td>
<td>settable</td>
<td>DMA1_Stream1</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
<td>settable</td>
<td>DMA1_Stream2</td>
</tr>
</tbody>
</table>
### STM32F4 vector table from startup code (partial)

<table>
<thead>
<tr>
<th>Vectors</th>
<th>DCD</th>
<th>__initial_sp</th>
<th>; Top of Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCD</td>
<td>Reset_Handler</td>
<td>; Reset Handler</td>
<td></td>
</tr>
<tr>
<td>DCD</td>
<td>NMI_Handler</td>
<td>; NMI Handler</td>
<td></td>
</tr>
</tbody>
</table>

......

| DCD     | SVC_Handler    | ; SVCall Handler |
| DCD     | DebugMon_Handler| ; Debug Monitor Handler |
| DCD     | 0              | ; Reserved |
| DCD     | PendSV_Handler | ; PendSV Handler |
| DCD     | SysTick_Handler| ; SysTick Handler |

; External Interrupts

| DCD     | WWDG_IRQHandler    | ; Window WatchDog |
| DCD     | PVD_IRQHandler     | ; PVD via EXTI Line detection |
| DCD     | TAMPP_STAMP_IRQHandler| ; Tamper/TimeStamps via EXTI |
| DCD     | RTC_WKUP_IRQHandler | ; RTC Wakeup via EXTI line |
| DCD     | FLASH_IRQHandler   | ; FLASH |
| DCD     | RCC_IRQHandler     | ; RCC |
| DCD     | EXTI0_IRQHandler   | ; EXTI Line0 |
| DCD     | EXTI1_IRQHandler   | ; EXTI Line1 |
| DCD     | EXTI2_IRQHandler   | ; EXTI Line2 |
Special CPU registers

ARM instructions to “access special registers”

MRS Rd,spec ; move from special register (other than R0-R15) to Rd
MSR spec,Rs ; move from register Rs to special register

Use CMSIS\(^1\) functions to clear/set PRIMASK

\_\_enable\_irq();  // enable interrupts (set PRIMASK=0)
\_\_disable\_irq();  // disable interrupts (set PRIMASK=1)

(double-underscore at beginning)

Special Cortex-M Assembly Language Instructions

CPSIE I ; Change Processor State/Enable Interrupts (sets PRIMASK = 0)
CPSID I ; Change Processor State/Disable Interrupts (sets PRIMASK = 1)

Prioritized Interrupts Mask Register (PRIMASK)

PRIMASK = 1 prevents (masks) activation of all exceptions with configurable priority
PRIMASK = 0 permits (enables) exceptions

Processor Status Register (PSR)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23</th>
<th>16 15</th>
<th>10 9 8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N  Z  C  V  Q  ICI/IT  T</td>
<td>Reserved</td>
<td>ICI/IT</td>
<td>ISR_NUMBER</td>
</tr>
</tbody>
</table>

1 Cortex Microcontroller Software Interface Standard – Functions for all ARM Cortex-M CPUs, defined in project header files: core\_cmFunc.h, core\_cm3.h
Prioritized interrupts

- Up to 256 priority levels
- 8-bit priority value
- Implementations may use fewer bits
  - STM32F4xx uses upper 4 bits of each priority byte -> 16 levels
  - STM32F4xx uses 4 bits -> 16 levels
- NMI & HardFault priorities are fixed
“Tail-chaining” interrupts

- NVIC does not unstack registers and then stack them again, if going directly to another ISR.
- NVIC can halt stacking (and remember its place) if a new IRQ is received.
Exception return

- The exception mechanism detects when the processor has completed an exception handler.
- Exception return occurs when:
  1. Processor is in Handler mode
  2. EXC_RETURN loaded to PC
  3. Processor executes one of these instructions:
     - LDM or POP that loads the PC
     - LDR with PC as the destination
     - BX using any register
- EXC_RETURN value loaded into LR on exception entry (after stacking original LR)
  - Lowest 5 bits of EXC_RETURN provide information on the return stack and processor mode.
Interrupt signal: from device to CPU

In each peripheral device:
- Each potential interrupt source has a separate **arm (enable)** bit
  - Set for devices from which interrupts are to be accepted
  - Clear to prevent the peripheral from interrupting the CPU
- Each potential interrupt source has a separate **flag** bit
  - Hardware sets the flag when an “event” occurs
  - Interrupt request = (flag & enable)
  - ISR software must clear the flag to acknowledge the request
  - test flags in software if interrupts not desired

Nested Vectored Interrupt Controller (NVIC)
- Receives all interrupt requests
- Each has an enable bit and a priority within the VIC
- Highest priority enabled interrupt sent to the CPU

Within the CPU:
- Global interrupt enable bit in PRIMASK register
- Interrupt if priority of IRQ < that of current thread
- Access interrupt vector table with IRQ#
Nested Vectored Interrupt Controller

- NVIC manages and prioritizes external interrupts in Cortex-M
  - 82 IRQ sources from STM32F4xx peripherals
- NVIC interrupts CPU with IRQ# of highest-priority IRQ signal
  - CPU uses IRQ# to access the vector table & get intr. handler start address
**NVIC registers (one bit for each IRQ#)**

- **NVIC_ISERx/NVIC_ICERx**
  - Each IRQ has its own **enable** bit within NVIC
  - Interrupt Set/Clear Enable Register
  - 1 = Set (enable) interrupt/Clear (disable) interrupt

- **NVIC_ISPRx/NVIC_ICPRx**
  - Interrupt Set/Clear Pending Register
  - Read 1 from ISPR if interrupt in pending state
  - Write 1 to set interrupt to pending or clear from pending state

- **NVIC_IABRx – Interrupt Active Bit Register**
  - Read 1 if interrupt in active state

\[ x = 0..7 \] for each register type, with 32 bits per register, to support up to 240 IRQs (82 in STM32F4xx)

- Each bit controls one interrupt, identified by its IRQ# (0..239)
- Register# \( x = \) IRQ# DIV 32
- Bit n in the register = IRQ# MOD 32
NVIC registers (continued)

- NVIC_IPRx (x=0..59) – Interrupt Priority Registers
  - Supports up to 240 interrupts: 0..239 (82 in STM32F4)
  - 8-bit priority field for each interrupts (4-bit field in STM32F4)
    - 4 priority values per register (STM32F4 – upper 4 bits of each byte)
    - 0 = highest priority
    - Register# x = IRQ# DIV 4
    - Byte offset within the register = IRQ# MOD 4
    - Ex. IRQ85:
      - 85/4 = 21 with remainder 1 (register 21, byte offset 1)
        - Write priority<<8 to NVIC_IPR2
      - 85/32 = 2 with remainder 21: write 1<<21 to NVIC_SER2

- STIR – Software Trigger Interrupt Register
  - Write IRQ# (0..239) to trigger that interrupt from software
  - Unprivileged access to this register enabled in system control register (SCR)
NVIC example (assembly language)

NVIC_ISER0/1/2 = 0xE000E100/104/108
NVIC_ICER0/1/2 = 0xE000E180/184/188
NVIC_IPR0/1/2/…/20 = 0xE00E400/404/408/40C/…./500

;Example – Enable EXTI0 with priority 5 (EXTI0 = IRQ6)
NVIC_ISER0 EQU 0xE000E100 ;bit 6 enables EXTI0
NVIC_IPR1 EQU 0xE000E404 ;3rd byte = EXTI0 priority

ldr r0,=NVIC_ISER0
mov r1,#0x0040 ;Set bit 6 of ISER0 for EXTI0
str r1,[r0]
ldr r0,=NVIC_IPR1 ;IRQ6 priority in IPR1[23:16]
ldr r1,[r0] ;Read IPR1
bic r1,#0x00FF0000 ;Clear [23:16] for IRQ6
orr r1,#0x00500000 ;Bits [23:20] = 5
str r1,[r0] ;Upper 4 bits of byte = priority
CMSIS: Cortex Microcontroller Software Interface Standard

Vendor-independent hardware abstraction layer for Cortex-M
(Facilitates software reuse)

- **Core Peripheral Access Layer** provides name definitions, address definitions, and helper functions to access core registers and core peripherals.
- **Device Peripheral Access Layer (MCU specific)** offers name definitions, address definitions, and driver code to access peripherals.
- **Access Functions for Peripherals (MCU specific and optional)** implements additional helper functions for peripherals.
CMSIS functions
Available when CMSIS Core is included in the project

- **NVC_EnableIRQ**(IRQn_Type IRQn)
- **NVIC_DisableIRQ**(IRQn_Type IRQn)
  - Set bit IRQn in NVIC_ISERx/NVIC_ICERx
- **NVIC_SetPendingIRQ**(IRQn_Type IRQn)
- **NVIC_ClearPendingIRQ**(IRQn_Type IRQn)
- **NVIC_GetPendingIRQ**(IRQn_Type IRQn)
  - Set/read/read bit IRQn in NVIC_ISPRx/NVIC_ICPRx
- **NVIC_SetPriority**(IRQn_Type IRQn, unit32_t priority)
- **NVIC_GetPriority**(IRQn_Type IRQn)
  - Set/get IRQn priority in NVIC_IPRxI
NVIC CMSIS example: enable interrupts

- **Interrupt Set Enable Register**: each bit *enables* one interrupt
  
  \[
  \text{NVIC\_EnableIRQ}(n); \quad // \text{set bit to enable IRQn}
  \]

- **Interrupt Clear Enable Register**: each bit *disables* one interrupt
  
  \[
  \text{NVIC\_DisableIRQ}(n); \quad // \text{set bit to disable IRQn}
  \]

- For convenience, *stm32f4xx.h* defines a symbol for each IRQn
  
  Examples:
  
  \[
  \text{EXTI0\_IRQn} = 6; \quad // \text{External interrupt EXTI0 is IRQ #6}
  \]
  
  \[
  \text{TIM3\_IRQn} = 29; \quad // \text{Timer TIM3 interrupt is IRQ #29}
  \]

  Usage:
  
  \[
  \text{NVIC\_EnableIRQ(EXTI0\_IRQn);} \quad // \text{enable external interrupt EXTI0}
  \]
  
  \[
  \text{NVIC\_DisableIRQ(TIM3\_IRQn);} \quad // \text{disable interrupt from timer TIM3}
  \]
STM32F4 external interrupt/event controller

23 edge detectors to trigger events and interrupts signaled by 240 GPIO pins and 7 internal events.
STM32F4xx external interrupt sources
(select in System Configuration Module – SYSCFG)

- 16 multiplexers select GPIO pins as external interrupts EXTI0..EXTI15
- Mux inputs selected via 4-bit fields of EXTICR[k] registers (k=0..3)
  - EXTIx = 0 selects PAx, 1 selects PBx, 2 selects PCx, etc.
  - EXTICR[0] selects EXTI3-EXTI0; EXTICR[1] selects EXTI7-EXTI4, etc

Example: Select pin PC2 as external interrupt EXTI2

```c
SYSCFG->EXTICR[0] &= 0xF0FF; //clear EXTI2 bit field
SYSCFG->EXTICR[0] |= 0x0200; //set EXTI2 = 2 to select PC2
```
STM32F4 external interrupt sources

**Sixteen external interrupts**
EXTI0 – EXTI15

**Seven “event” triggers:**
EXTI16 = PVD output
EXTI17 = RTC Alarm event
EXTI18 = USB OTG FS Wakeup event
EXTI19 = Ethernet Wakeup event
EXTI20 = USB OTG HS Wakeup event
EXTI21 = RTC Tamper and TimeStamp events
EXTI22 = RTC Wakeup event
STM32F4 EXTI Registers

23 bits per register - control 23 interrupts/events

- EXTI_IMR – interrupt mask register
  - 0 masks (disables) the interrupt
  - 1 unmaskes (enables) the interrupt

- EXTI_RTSR/FTSR – rising/falling trigger selection register
  - 1 to enable rising/falling edge to trigger the interrupt/event
  - 0 to ignore the rising/falling edge

- EXTI_PR – interrupt/event pending register
  - read 1 if interrupt/event occurred
  - clear bit by writing 1 (writing 0 has no effect)
  - write 1 to this bit in the interrupt handler to clear the pending state of the interrupt

- EXTI_SWIER – software interrupt event register
  - 1 to set the pending bit in the PR register
  - Triggers interrupt if not masked
Signal flow/setup for External Interrupt EXTI\(x\), \(x = 0 \ldots 15\)

**SYSCFG module**

- Select source for EXTI\(x\)
  - 4 bits each
    - 0000 = PAx
    - 0001 = PBx
    - 0010 = PCx
    - 0011 = PDx
    - ... (continued)
  - SYSCFG -> EXICR[0]

**GPIO module**

- Configure PAx as input (00)
- GPIOA -> MODER
- Pin PAx

**EXTI module**

- For all EXTI\(x\) registers: Bit \(n\) controls EXTI\(n\)
- EXTI \(\rightarrow\) IMR
- EXTI \(\rightarrow\) FTSR
- EXTI \(\rightarrow\) RTSR

**CPU**

- Enable
- IRQ
- 
  - 
  - enable_irq();
  - disable_irq();

**NVIC**

- NVIC_EnableIRQ(n); interrupt mask (enable)
- Pending flag
  - Set when EXTI\(x\) activates Clear when intr. handler exits
  - NVIC_ClearPendingIRQ(n); clear pending flag
  - NVIC_SetPriority(intr. #, value); value = priority of intr. #

- EXTI \(\rightarrow\) PR
- EXTI \(\rightarrow\) edge
  - Set by HW
  - Clear by SW
  - enable this interrupt
  - select falling and/or rising edge

1 of 45, passed by NVIC to CPU
Project setup for interrupt-driven applications

- Write the interrupt handler for each peripheral
  - Clear the flag that requested the interrupt (acknowledge the intr. request)
  - Perform the desired actions, communicating with other functions via shared global variables
  - Use function names from the vector table
    - Example: `void EXTI4_IRQHandler () { statements }`

- Perform all initialization for each peripheral device:
  - Initialize the device, “arm” its interrupt, and clear its “flag”
    - Example: External interrupt EXTI
      - Configure GPIO pin as a digital input
      - Select the pin as the EXTI source (in SYSCFG module)
      - Enable interrupt to be requested when a flag is set by the desired event (rising/falling edge)
      - Clear the pending flag (to ignore any previous events)

- NVIC
  - Enable interrupt: `NVIC_EnableIRQ (IRQn);`
  - Set priority: `NVIC_SetPriority (IRQn, priority);`
  - Clear pending status: `NVIC_ClearPendingIRQ (IRQn);`

- Initialize counters, pointers, global variables, etc.
- Enable CPU Interrupts: `__enable_irq () ;`

(diagram on next slide)
Example: Enable EXTI0 as rising-edge triggered

;System Configuration Registers
SYSCFG    EQU     0x40013800
EXTICR1   EQU     0x08

;External Interrupt Registers
EXTI      EQU     0x40013C00
IMR       EQU     0x00    ;Interrupt Mask Register
RTSR      EQU     0x08    ;Rising Trigger Select
FTSR      EQU     0x0C    ;Falling Trigger Select
PR        EQU     0x14    ;Pending Register

;select PC0 as EXTI0
ldr    r1,=SYSCFG    ;SYSCFG selects EXTI sources
ldrh   r2,[r1,#EXTICR1] ;EXTICR1 = sources for EXTI0 - EXTI3
bic    r2,#0x000f    ;Clear EXTICR1[3-0] for EXTI0 source
orr    r2,#0x0002    ;EXTICR1[3-0] = 2 to select PC0 as EXTI0 source
strh   r2,[r1,#EXTICR1] ;Write to select PC0 as EXTI0

;configure EXTI0 as rising-edge triggered
ldr    r1,=EXTI      ;EXTI register block
mov    r2,#1         ;bit #0 for EXTI0 in each of the following registers
str    r2,[r1,#RTSR] ;Select rising-edge trigger for EXTI0
str    r2,[r1,#PR]   ;Clear any pending event on EXTI0
str    r2,[r1,#IMR]  ;Enable EXTI0
Interrupt Handler – count button presses
void EXTI0_IRQHandler(void) {
    if (!(GPIOA->IDR & (1<<0)) ) {
        count++;
    }
    NVIC_ClearPendingIRQ(EXTI3_IRQn);
    EXTI->PR|=(1<<0);
}

Initialize the GPIO and the external interrupt
void Init_Switch(void){
    RCC->AHB1ENR|= RCC_AHB1ENR_GPIOAEN;
    GPIOA->PUPDR |= GPIO_PUPDR_PUPDR0_1;
    SYSCFG->EXTICR[0] &= SYSCFG_EXTICR1_EXTI0_PA;
    EXTI->IMR |= (1<<0);
    EXTI->FTSR |= (1<<0);
    __enable_irq();
    NVIC_SetPriority(EXTI0_IRQn,0);
    NVIC_ClearPendingIRQ(EXTI0_IRQn);
    NVIC_EnableIRQ(EXTI0_IRQn);
}

Interrupt Handler – count button presses
void EXTI0_IRQHandler(void) {
    if (!(GPIOA->IDR & (1<<0)) ) {
        count++;
    }
    NVIC_ClearPendingIRQ(EXTI3_IRQn);
    EXTI->PR|=(1<<0);
}
Supervisor Call Instruction (SVC)

- Access system resources from O/S (“privileged operations”)
- **SVC_Handler** is defined in the interrupt vector table
- SVC interrupt handler written as a C function:
  ```c
  void SVC_Handler()
  {
    your code
  }
  ```
- SVC interrupt handler as an assembly language function:
  ```assembly
  EXTERN SVC_Handler
  SVC_Handler
  your code
  bx lr
  ```
Supervisor Call instruction (SVC)

- To execute \texttt{SVC\_Handler} as a software interrupt
  - Assembly language syntax: \texttt{SVC \#imm}
  - C syntax: \texttt{\_\_svc (imm)}

- \texttt{imm} is an “SVC number” (0-255), which indicates a particular “service” to be performed by the handler
  - \texttt{imm} is encoded into the instruction, but ignored by the CPU
  - Handler can retrieve \texttt{imm} by using stacked PC to read the SVC instruction code (examples provided later)

- Since this is an “interrupt”, R0-R3 are pushed onto the stack:
  - Arguments can be passed to the handler in R0-R3
  - SVC handler can retrieve the arguments from the stack
  - SVC handler can also return results by replacing R0-R3 values in the stack, which will be restored to R0-R3 on return from interrupt.
Access SVC arguments in C

// Stack contains eight 32-bit values:
//   r0, r1, r2, r3, r12, r14, return address, xPSR
// 1st argument = r0 = svc_args[0]
// 2nd argument = r1 = svc_args[1]
// 7th argument = return address = svc_args[6]

void SVC_Handler(unsigned int * svc_args) {
    int a, b, c;

    a = svc_args[0]; //get first argument from stack
    b = svc_args[1]; //get second argument from stack
    c = a + b;
    svc_args[0] = c; //replace R0 value in stack with result to “return” result in R0
}

Access SVC arguments in assembly language

; Stack contains: r0, r1, r2, r3, r12, r14, return address, xPSR
; The saved r0 is the top entry in the stack

EXPORT SVC_Handler

SVC_Handler

TST    LR,#0x04 ; Test bit 2 of EXC_RETURN
ITE    EQ ; Which stack pointer was used?
MRSEQ  R4,MSP ; Copy Main SP to R4
MRSNE  R4,PSP ; Copy Process SP to R4
LDR    R1,[R4] ; Retrieve saved R0 from top of stack
LDR    R2,[R4,#4] ; Retrieve saved R1 from stack
....
STR    R1,[R4] ; Replace saved R0 value in stack
BX     LR ; Return and restore registers from stack
SVC in C programs

- May associate “standard” function name with the `__svc (imm)` function call
  - May pass up to four integer arguments
  - May return up to four results in a “value_inRegs” structure
  - Syntax:
    ```
    __svc(int svc_num) return-type function-name(argument-list)
    ```
    - `svc_num` (8-bit constant) = immediate value in SVC instruction
    - “return-type function-name(argument-list)” = C function prototype
- Call the function via: `function-name(argument-list);`
  (examples on next slide)
Example: SVC call from C code

/*----------------------------------------------------------
   Set up SVC “calls” to “SVC_Handler”
   SVC_Handler function must be defined elsewhere
/*----------------------------------------------------------*/
#define SVC_00 0x00
#define SVC_01 0x01

/* define function “svc_zero” as SVC #0, passing pointer in R0 */
/* define function “svc_one” as SVC #1, passing pointer in R0 */
void __svc(SVC_00)    svc_zero(const char *string);
void __svc(SVC_01)    svc_one(const char *string);

int call_system_func(void) {
    svc_zero("String to pass to SVC handler zero");            //Execute SVC #0
    svc_one("String to pass to a different OS function");     //Execute SVC #1
}

Reference: ARM Compiler toolchain Developing Software for
            ARM Processors": Supervisor Calls, Example 56
void SVC_Handler(unsigned int * svc_args) {
  unsigned int svc_number;

  // Read SVC# byte from SVC instruction code
  svc_number = ((char *)svc_args[6])[-2];

  // Execute code for each SVC #
  switch(svc_number) {
    case SVC_00: /* Handle SVC 00 */
      break;
    case SVC_01: /* Handle SVC 01 */
      break;
    default: /* Unknown SVC */
      break;
  }
}
Access SVC immediate operand in assembly language

; Parameters in R0-R3 were pushed onto the stack

EXPORT SVC_Handler

SVC_Handler

TST   LR,#0x04 ; Test bit 2 of EXC_RETURN
ITE   EQ   ; Which stack pointer was used?
MRSEQ R0,MSP ; Copy Main SP to R0
MRSNE R0,PSP ; Copy Process SP to R0
LDR   R1,[R0,#24] ; Retrieve stacked PC from stack
LDRB  R0,[R1,#-2] ; Get #N from SVC instruction in program
ADR   R1,SVC_Table ; SVC Vector Table address
LDR   PC,[R1,R0,SLL #2] ; Branch to Nth routine

....

SVC_TABLE ; Table of function addresses

DCD  SVC0_Function
DCD  SVC1_Function
DCD  SVC2_Function
System tick timer interrupts

- **SysTick Timer** is a simple 24-bit down counter
  - Interrupt on count down from 1 -> 0
  - Counter rolls over from 0 to 24-bit “reload” value (determines interrupt period)
  - User provides interrupt handler: `SysTick_Handler(void)`

- Control register bits:
  - 0: enable
  - 1: interrupt enable
  - 2: clock source
    - FCLK = free-running internal core clock (default)
    - STCLK = external clock signal
  - 16: rollover flag (set on count down from 1->0)

- CMSIS function starts timer, enables interrupt, selects clock source and sets reload value:

  ```c
  #include "core_cm4.h"
  SysTick_Config (numberOfTicks);  //Ex. #ticks = SystemCoreClock/1000
  ```